OpenGL Functions

Initializing, Creating, and Destroying a Window

void **glutInit**(int *argc*, char ***argv*);

Initializes GLUT and processes any command-line arguments. The command line options are dependent on the window system. **glutInit()** should be called before any other GLUT routine.

void **glutInitDisplayMode**(unsigned int *mode*);

Specifies a display mode for windows created when **glutCreateWindow()** is called. The mask argument is a bitwise Ored combination of GLUT_RGBA or GLUT_INDEX, GLUT_SINGLE or GLUT_DOUBLE, and any of the buffer-enabling flags: GLUT_DEPTH, GLUT_STENCIL, or GLUT_ACCUM. The default value is GLUT_RGBA | GLUT_SINGLE.

void **glutInitWindowSize**(int *width*, int *height*);

Specifies the *width* and *height*, in pixels, of your window. The initial window size is only a hint and may be overridden by other requests.

void **glutInitWindowPosition**(int *x*, int *y)*;

Specifies the initial *x* and *y* location for the upper-left corner of the window in relation to the upper-left corner of the monitor's screen. The initial window position is only a hint and may be overridden by other requests.

int **glutCreateWindow**(char **name*);

Creates a window with an OpenGL context using the previously set characteristics (display mode, width, height, etc.) The string *name* appears in the title bar. The window is not initially displayed until **glutMainLoop()** is entered; consequently, no rendering should be done until then. The value returned is a unique identifier for the window. This identifier can be used for controlling and rendering to multiple windows (each with an OpenGL rendering context) from the same application.

int **glutSetWindow**(int *windowID*);

Sets *windowID* as the current window.

int **glutGetWindow**(void);

Returns the current window ID or zero if no windows exist or current window was destroyed.

int **glutDestroyWindow**(int *windowID*);

Destroys the window having *windowID* and also any of its subwindows.

int **glutCreateSubWindow**(int *windowID*, int *x*, int *y*, int *width*, int *height*);

Creates a subwindow of the window identified by *windowID* of size *width* and *height* at location *x* and *y* within the current window. Implicitly, the current window is set to the newly created subwindow. The display state of a window is initially for the window to be shown. However, the window's display state is not actually acted upon until **glutMainLoop()** is entered. This means until **glutMainLoop()** is called, rendering to a created window is ineffective. Subwindows cannot be iconified. Subwindows can be nested arbitrarily deep. The value returned by the **glutCreateSubWindow()** call is a unique small integer identifier for the window starting with one.

Running the Program

void **glutMainLoop**(void);

Enters the GLUT processing loop, which is an infinite loop. Registered callback functions are called whenever the corresponding event occurs.

Viewport Transformation

void **glViewPort**(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

Defines a pixel rectangle in the window into which the final image is mapped. The (*x*, *y*) parameter specifies the lower left corner of the viewport, and *width* and *height* are the size of the viewport rectangle. By default, the initial viewport values are (0,0, *winWidth*, *winHeight*), where *winWidth* and *winHeight* specify the size of the window.

void **glDepthRange**(GLclamped *near*, GLclamped *far*);

Defines an encoding for z-coordinates that's performed during the viewport transformation. The *near* and *far* values represent adjustments to the minimum and maximum values that can be stored in the depth buffer. By default, they are 0.0 and 1.0, respectively, which work for most applications. These parameters are clamped to lie within [0, 1].

Attribute Groups

void **glPushAttrib**(GLbitfield *mask*); void **glPopAttrib**(void);

glPushAttrib() saves all the attributes indicated by bits in *mask* by pushing them onto the attribute stack. **glPopAttrib()** restores the values of those state variables that were saved with the last call to **glPushAttrib()**. Table 2-6 lists the possible mask bits that can be logically ORed together to save any combination of attributes. The special mask GL_ATTRIB_BITS is used to save and restore all the state variables in all the attribute groups. GL_CURRENT_BIT saves the current color, texture coordinates, normal, etc.

void **glPushClientAttrib**(GLbitfield *mask*); void **glPopClientAttrib**(void);

glPushClientAttrib() saves all the attributes indicated by bits in *mask* by pushing them onto the client attribute stack. **glPopClientAttrib()** restores the values of those state variables that were saved with the last call to **glPushClientAttrib()**. Table 2-7 lists the possible mask bits that can be logically ORed together to save any combination of attributes.

Manipulating the Matrix Stacks

void **glPushMatrix**(void);

Pushes all matrices in the current stack down one level. The current stack is determined by **glMatrixMode()**. The topmost matrix is copied, so its contents are duplicted in both the top and second-from-the-top matrix. If too many matrices are pushed, an error is generated.

void **glPopMatrix**(void);

Pops the top matrix off the stack, destroying the contents of the popped matrix. The matrix that was second from the top becomes the top matrix. The current stack is determined by the **glMatrixMode()**. If the stack contains a single matrix, calling **glPopMatrix()** generates an error.

Basic Drawing Commands

void **glClearColor**(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);

Sets the current clearing color for use in clearing color buffers in RGBA mode.

void **glClear**(GLbitfield *mask*);

Clears the specified buffers to their current clearing values. The mask argument is a bitwise logical OR combination of the values listed in Table 2-1.

void **glFlush**(void);

Forces previously issued OpenGL commands to begin execution, thus guaranteeing that they complete.

void **glFinish**(void);

Forces all previously issued OpenGL commands to complete.

Describing Points, Lines, and Polygons

void **glVertex**{234}{sifd}[v](TYPE *coords*);

Specifies a vertex for use in describing a geometric object. Use only between a **glBegin()** and **glEnd()** pair.

void **glBegin**(GLenum *mode*);

Marks the beginning of a vertex-data list that describes a geometric primitive.

Value	Meaning
GL_POINTS	Individual points
GL_LINES	Pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	Series of connected line segments
GL_LINE_LOOP	Same as above, with a segment added between last and first vertices
GL_TRIANGLES	Triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	Linked strips of triangles
GL_TRIANGLE_FAN	Linked fan of triangles
GL_QUADS	Quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	Linked strip of quadrilaterals
GL_POLYGON	Boundary of a simple, convex polygon

void **glEnd**(*void*);

Marks the end of a vertex-data list.

void **glEnable**(GLenum *cap*); void **glDisable**(GLenum *cap*);

Turns a capability on or off. There are over 40 enumerated values that can be passed as parameters.

Displaying Points, Lines, and Polygons

void **glPointSize**(GLfloat *size*);

Sets the width in pixels for rendered points; *size* must be > 0.0 and by default is 1.0.

void **glLineWidth**(GLfloat *width*);

Sets the width in pixels for rendered lines; *width* must be > 0.0 and by default is 1.0.

void **glLineStipple**(GLint *factor*, GLushort *pattern*);

Sets the current stippling pattern for lines. The *pattern* argument is a 16-bit series of 0s and 1s, and is repeated as necessary to stipple a given line. The *factor* argument multiplies each subseries of consecutive 1s and 0s. Line stippling must be enabled by passing GL_LINE_STIPPLE to **glEnable()**.

void **glPolygonMode**(GLenum *face*, GLenum *mode*);

Controls the drawing mode for a polygon's front and back faces. The parameter face can be GL_FRONT_AND_BACK, GL_FRONT, or GL_BACK; mode can be GL_POINT, GL_LINE, or GL_FILL to indicate whether the polygon should be drawn as points, outlined, or filled. By default, both the front and back faces are drawn filled. For example, the following example has the front faces filled and the back faces outlined.

```
glPolygonMode(GL_FRONT,
GL_FILL);
glPolygonMode(GL_BACK,
GL_LINE);
```

void **glFrontFace**(GLenum *mode*);

Controls how front-facing polygons are determined. By default, mode is GL_CCW. If mode is GL_CW, faces with clockwise rotation are considered front facing.

void **glCullFace**(GLenum *mode*);

Indicates which polygons should be discarded (culled) before they are converted to screen coordinates. The mode is either GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK. To take effect, culling must be enabled using **glEnable()** with GL_CULL_FACE.

void **glPolygonStipple**(const GLubyte ***mask*);

Defines the current stipple pattern for polygons. The argument mask is a pointer to a 32 x 32 bitmap that is interpreted as a mask of 0s and 1s.

void **glEdgeFlag**(GLboolean *flag*);
void **glEdgeFlagv**(const GLboolean ***flag*);

Indicates whether a vertex should be considered as initializing a boundary edge of a polygon. If *flag* if GL_TRUE, the edge flag is set to TRUE (the default), and any vertices created are considered to precede boundary edges until this function is called again with *flag* being GL_FALSE.

void **glNormal3**{bsidf}(TYPE *nx*, TYPE *ny*, TYPE *nz*); void **glNormal3**{bsidf}v(const TYPE *v);

Sets the current normal vector as specified by the arguments.

General-Purpose Transformation Commands

void **glMatrixMode**(GLenum *mode*);

Specifies whether the model view, projection, or texture matrix will be modified, using the argument GL_MODELVIEW, GL_P ROJECTION, or GL_TEXTURE for *mode*. Subsequent transformation commands affect the specified matrix. Note that only one matrix can be modified at a time.

void **glLoadIdentity**(void);

Sets the currently modifiable matrix to the 4 x 4 identity matrix.

void **glLoadMatrix**{fd}(const TYPE ***m*);

Sets the 16 values of the current matrix to those specified by *m*, which should be declared as m[16].

void **glMultiMatrix**{fd}(const TYPE *_m_);

Multiplies the matrix specified by the 16 values pointed to by *m* by the current matrix and stores the result as the current matrix. The order of multiplication is CM, where C is the current matrix on the stack.

Viewing and Modeling Transformations

void **glTranslate**{fd}(TYPE *x*, TYPE *y*, TYPE *z*);

Multiplies the current matrix by a matrix that moves (translates) an object by the given *x*-, *y*-, and z-values (or moves the local coordinate system by the same amount).

void **glRotate**{fd}(TYPE *angle*, TYPE *x*, TYPE *y*, TYPE *z*);

Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (*x*, *y*, *z*). The *angle* parameter specifies the angle of rotation in degrees.

void **glScale**{fd}(TYPE *x*, TYPE *y*, TYPE *z*);

Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each *x*-, *y*-, and *z*-coordinate of every point in the object is multiplied by the corresponding argument *x*, *y*, or *z*. With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or relected by the *x*-, *y*-, and *z*-factors, and the associated object is transformed with them.

void **gluLookAt**(GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*,
GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*, GLdouble *upx*, GLdouble *upy*, GLdouble *upz*);

Defines a viewing matrix and multiplies it to the right of the current matrix. The desired viewpoint is specified by *eyex*, *eyey*, and *eyez*. The *centerx*, *centery*, and *centerz* arguments specify any point along the desired line of sight, but typically they specify some point in the center of the scene being looked at. The *upx*, *upy*, and *upz* arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume).

Projection Transformations

void **glFrustum**(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Creates a matrix for a perspective-view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the following parameters: (*left*, *bottom*, *-near*) and (*right*, *top*, *-near*) specify the (x, y, z) coordinates of the lower left and upper right corners, respectively, of the near clipping plane; *near* and *far* give the distances from the viewpoint to the near and far clipping planes. They should always be positive. The frustum has a default orientation in 3D space. You can perform rotations or translations on the projection matrix to alter this orientation, but this is tricky and nearly always avoidable.

void **gluPerspective**(GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*, GLdouble *far*);

Creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. *fovy* is the angle of the field of view in the yz-plane; its value must be in the range [0.0, 180.0]. *aspect* is the aspect ratio of the frustum (i.e., its width divided by its height). *near* and *far* values are the distances between the viewpoint and the clipping planes along the negative z-axis. They should always be positive. Just as with **glFrustum()**, you can apply rotations or translations to change the default orientation of the viewing volume created by **gluPerspective()**. With no such transformations, the viewpoint remains at the origin, and the line of sight points down the negative z-axis.

void **glOrtho**(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (*left*, *bottom*, *-near*) and (*right*, *top*, *-near*) are points on the near clipping plane that are mapped to the lower left and upper right corners of the viewport window, respectively. (*left*, *bottom*, *-far*) and (*right*, *top*, *-far*) are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both *near* and *far* may be positive, negative, or even set to zero. However, near and far should not be the same value. With no other transformations, the direction of projection is parallel to the z-axis, and the viewpoint faces towards the negative z-axis.

void **gluOrtho2D**(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*);

Creates a matrix for projecting two-dimensional coordinates onto the screen and multiplies the current project matrix by it. The clipping region is a rectangle with the lower left corner at (*left*, *bottom*) and the upper right corner at (*right*, *top*). The z coordinates are assumed to lie between –1.0 and 1.0. In two-dimensional objects, all z coordinates are zero; thus, none of the objects are clipped because of its z-value.

void **glClipPlane**(GLenum *plane*, const GLdouble **equation*);

Defines a clipping plane. The *equation* argument points to the four coefficients of the plane equation, Ax + By +Cz + D = 0. The plane argument is GL_CLIP_PLANE*i*, where *i* is an integer specifying which one of the available clipping planes to define. The value of *i* begins at zero.

int **gluUnProject**(GLdouble *winx*, GLdouble *winy*, GLdouble *winz*, const GLdouble *modelviewMatrix*[16], const GLdouble *projectionMatrix*[16], const GLint *viewport*[4], GLdouble **objx*, GLdouble **objy*, GLdouble **objz*);

Maps the specified window coordinates (winx, winy, winz) into object coordinates, using transformations defined by a *modelviewMatrix*, *projectionMatrix*, and *viewport*. The resulting object coordinates are returned in *objx*, *objy*, and *objz*. The function returns GL_TRUE, indicating success, or GL_FALSE, indicating failure (such as a noninvertible matrix). This operation does not attempt to clip the coordinates to the viewport or eliminate depth values that fall outside of glDepthRange(). By default, a *winz* value of 0.0 will request the world coordinates of the transformed point at the near clipping plane, while a *winz* value of 1.0 will request the point at the far clipping plane. Use the following to get the three matrices:

```
glGetIntegerv(GL_VIEWPORT, viewport);
glGetDoublev(GL_MODELVIEW_MATRIX,
modelviewMatrix);
glGetDoublev(GL_PROJECTION_MATRIX,
projectionMatrix);
```

int **gluProject**(GLdouble *objx*, GLdouble *objy*, GLdouble *objz*, const GLdouble *modelviewMatrix*[16], const GLdouble *projectionMatrix*[16], const GLint *viewport*[4], GLdouble **winx*, GLdouble **winy*, GLdouble **winz*);

Maps the specified object coordinates (*objx*, *objy*, and *objz*) into window coordinates, using transformations defined by *modelviewMatrix*, *projectionMatrix*, and *viewport*. The resulting window coordinates are returned in *winx*, *winy*, and *winz*. The function returns GL_TRUE, indicating success, or GL_FALSE, indicating failure. **gluProject()** mimics the action of the transformation pipeline. Given 3D world coordinates and all the transformations that affect them, **gluProject()** returns the transformed window coordinates.

Color and Shading

void **glColor3**{bsifd ub us ui}(TYPE *r*, TYPE *g*, TYPE *b*);
void **glColor4**{bsifd ub us ui}(TYPE *r*, TYPE *g*, TYPE *b*, TYPE *a*);
void **glColor3**{bsifd ub us ui}v(const TYPE **v*);
void **glColor4**{bsifd ub us ui}v(const TYPE **v*);

Sets the current red, green, blue, and alpha values. The default alpha value is 1.0. The *v* indicates the argument is a pointer to an array of values. The typical range for the rgb values is [0, 1].

void **glShadeModel**(GLenum *mode*);

Sets the shading model. *mode* can be either GL_FLAT (the default) or GL_SMOOTH

(Gouraud shading).

Lighting

void **glLight**{if}(GLenum *light*, GLenum *pname*, TYPE *param*);
void **glLight**{if}v(GLenum *light*, GLenum *pname*, TYPE **param*);

Creates the light specified by light, which can by GL_LIGHT0, GL_LIGHT1, …, or GL_LIGHT7. The characteristic of the light being set is defined by *pname*, which specifies a parameter (see Table 5-1). *param* indicates the values to which the *pname* characteristic is set; it is a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can only be used to set single-valued light characteristics. Use **glEnable**(GL_LIGHTING) to enable lighting. Individual lights are turned on an off using **glEnable**(GL_LIGHT*i*) and **glDisable**(GL_LIGHT*i*), where *i* is a value in the range [0, 7]. The following call to **glClear()** is needed for lighting.

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

void **glLightModel**{if}(GLenum *pname*, TYPE *param*);
void **glLightModel**{if}v(GLenum *pname*, TYPE *param);

Sets properties of the lighting model. The characteristic of the lighting model being set is defined by pname, which specifies a names parameter (shown below). Param indicates the values to which the pnamecharacteristic is set; it is a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can only be used to set single-valued light model characteristics, not for GL_LIGHT_MODEL_AMBIENT.

Parameter Name	Default Value
GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0 or GL_FALSE
GL_LIGHT_MODEL_TWO_SIDE	0.0 or GL_FALSE
GL_LIGHT_MODEL_COLOR_CONTROL	GL_SINGLE_COLOR

Material Properties

void **glMaterial**{if}(GLenum *face*, GLenum *pname*, TYPE *param*);
void **glMaterial**{if}v(GLenum *face*, GLenum *pname*, TYPE **param*);

Specifies a current material property for use in lighting calculations. *face* can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate to which faces of the object the material should be applied. The particular material property being set is identified by *pname*, and the desired values for that property are given by *param*, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector version is used). The nonvector version only works for setting GL_SHININESS. The possible values for *pname* are shown in below. Note that GL_AMBIENT_AND_DIFFUSE allows you to set both the ambient and diffuse material colors simultaneously to the same RGBA value.

Parameter Name	Default Value
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)
GL_AMBIENT_AND_DIFFUSE	
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)
GL_SHININESS	0.0
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)
GL_COLOR_INDEXES	(0, 1, 1)

void **glColorMaterial**(GLenum *face*, GLenum *mode*);

Causes the material property (or properties) specified by *mode* of the specified material *face* (or faces) specified by face to track the value of the current color at all times. A change to the current color (using **glColor*()**) immediately updates the specified material properties. The *face* parameter can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK (the default). The *mode* parameter can be GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_AMBIENT_AND_DIFFUSE (the default), or GL_EMISSION. At any given time, only one mode is active. **glColorMaterial()** has no effect on color-index lighting.

Display Lists

GLuint **glGenLists**(GLsizei *range*);

Allocates *range* number of contiguous, previously unallocated display list indices. The integer returned is the index that marks the beginning of a contiguous block of empty display list indices. The returned indices are all marked as empty and used, so subsequent calls to **glGenLists()** don't return these indices until they are deleted.
Zero is returned if the requested number of indices isn't available, or if *range* is zero.

void **glNewList**(GLuint *list*, GLenum *mode*);

Specifies the start of a display list. OpenGL routines that are called subsequently (until **glEndList()** is called to end the display list) are stored is a display list, except for a few restricted OpenGL routines that can't be stored. *list* is a nonzero positive integer that uniquely identifies the

display list. The possible values for *mode* are GL_COMPILE and GL_COMPILE_AND_EXECUTE. Use GL_COMPILE if you do not want the OpenGL commands executed as they are placed in the display list; to cause the commands to be executed immediately as well as placed in the display list for later use, specify GL_COMPILE_AND_EXECUTE.

void **glEndList**(void);

Marks the end of a display list.

void **glCallList**(GLuint *list*);

Executes the display list specified by *list*. The commands in the display list are executed in the order they were saved, just as if they were issued without using a display list. If *list* hasn't been defined, nothing happens.

GLboolean **glIsList**(GLuint *list*);

Returns GL_TRUE if *list* is already used for a display list and GL_FALSE otherwise.

void **glDeleteLists**(GLuint *list*, GLsizei *range*);

Deletes *range* display lists, starting at the index specified by *list*. An attempt to delete a list that has never been created is ignored.

void **glListBase**(GLuint *base*);

Specifies the offset that is added to the display list indices in **glCallLists()** to obtain the final display list indices. The default display list *base* is zero. The list base has no effect on **glCallList()**, which executes only one display list, or on **glNewList()**.

void **glCallLists**(GLsizei *n*, GLenum *type*, const GLvoid **lists*);

Executes *n* display lists. The indices of the lists to be executed are computed by adding the offset indicated by the current display list base (specified with **glListBase()** routine) to the signed integer values in the array pointed to by *lists*. The *type* parameter indicates the data type of the of the values in list

Handling Window and Input Events

void **glutDisplayFunc**(void (***func*)(void));

Specifies the function *func* that is called whenever the contents of the window need to be redrawn. This may happen when the window is initially opened, when the window is popped and window damage is exposed, and when **glutPostRedisplay()** is explicitly called.

void **glutReshapeFunc**(void (***func*)(int *width*, int *height*));

Specifies the function that is called whenever the window is resized or moved. The argument func is a pointer to a function that expects two arguments, the new width and height of the window. Typically *func* calls **glViewport()**, so that the display is clipped to the new size, and it redefines the projection matrix so that the aspect ratio of the projected image matches the viewport, avoiding aspect ratio distortion. If glutReshapeFunct() is not called or is deregistered by passing NULL as *func*, a default reshape function is called, which calls glViewport(0, 0, *width*, *height*).

void **glutKeyboardFunc**(void (***func*)(unsigned char *key*, int *x*, int *y*));

Specifies the function *func* that is called when a key that generates an ASCII character is pressed. The *key* callback parameter is the generated ASCII value. The *x* and *y* callback parameters indicate the location (in window-relative coordinates) of the mouse pointer when the key was pressed.

void **glutSpecialFunc**(void (***func*)(unsigned char *key*, int *x*, int *y*));

Specifies the function *func* that is called whenever a special key such as an arrow key or a function key is pressed. The *key* callback parameter for an arrow key may be GLUT_KEY_UP, GLUT_KEY_DOWN, GLUT_KEY_LEFT, or GLUT_KEY_RIGHT. The *key* callback parameter for a function key may be GLUT_KEY_F*i*, where *i* is in the range [1,12]. The *x* and *y* callback parameters indicate the location (in window-relative coordinates) of the mouse pointer when the special key was pressed.

void **glutMouseFunc**(void (***func*)(int *button*, int *state*, int *x*, int *y*));

Specifies the function *func* that is called when a mouse button is pressed or released. The *button* callback parameter is GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. The *state* callback parameter is either GLUT_UP or GLUT_DOWN, depending on whether the mouse button has been released or pressed. The *x* and *y* callback parameters indicate the location (in window-relative coordinates) of the mouse pointer when the event occurred.

void **glutMotionFunc**(void (***func*)(int *x*, int *y*));

Specifies the function *func* that is called when the mouse pointer moves within the window while one or more mouse buttons are pressed. The x and y callback parameters indicate the location (in window-relative coordinates) of the mouse pointer when the event occurred.

void **glutPostRedisplay**(void);

Marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by **glutDisplayFunc()** will be called.

void **glutIdleFunc**(void (***func*)(void));

Specifies the function *func* to be executed if no other events are pending. If NULL is passed in , execution of *func* is disabled.

Drawing Three-Dimensional Objects

Note: All objects are rendered in immediate mode. Only the teapot generates texture coordinates.

void **glutWireSphere**(GLdouble *radius*, GLint *slices*, GLint *stacks*); void **glutSolidSphere**(GLdouble *radius*, GLint *slices*, GLint *stacks*);

void **glutWireCube**(GLdouble *size*); void **glutSolidCube**(GLdouble *size*);

void **glutWireCone**(GLdouble *radius*, GLdouble *height*, GLint *slices*, GLint *stacks*); void **glutSolidCone**(GLdouble *radius*, GLdouble *height*, GLint *slices*, GLint *stacks*);

void **glutWireTeapot**(GLdouble *size*); void **glutSolidTeapot**(GLdouble *size*);

void **glutWireTorus**(GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *nsides*, GLint *rings*); // A donut-shaped polyhedron
void **glutSolidTorus**(GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *nsides*, GLint *rings*);

void **glutWireTetrahedron**(void); // Polyhedron having 4 faces (i.e., a pyramid) void **glutSolidTetrahedron**(void);

void **glutWireOctahedron**(void); // Polyhedron having 8 faces void **glutSolidOctahedron**(void);

void **glutWireDodecahedron**(GLdouble radius); // Polyhedron having 12 faces void **glutSolidDodecahedron**(GLdouble radius);

void **glutWireIcosahedron**(void); // Polyhedron having 20 faces void **glutSolidIcosahedron**(void);

OpenGL functions for Hidden surface removal

Back face culling:

OpenGL face culling calculates the signed area of the filled primitive in window coordinate space. The signed area is positive when the window coordinates are in a counter-clockwise order and negative when clockwise. Back face culling technique in openGL is performed by using glFrontFace(). to specify the ordering, counter-clockwise or clockwise, to be interpreted as a front-facing or back-facing primitive

Syntax:
void **glFrontFace**(GLenum *mode*);

Parameters
mode
Specifies the orientation of front-facing polygons. GL_CW and GL_CCW are accepted. The initial value is GL_CCW.
An application can specify culling either front or back faces by calling glCullFace() .
Syntax:
void **glCullFace**(GLenum *mode*);

Parameters
mode
Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK are accepted. The initial value is GL_BACK. Finally, face culling must be enabled with a call to glEnable(GL_CULL_FACE);.

Syntax:
void **glEnable**(GLenum *cap*);
void **glDisable**(GLenum *cap*);
void **glEnablei**(GLenum *cap*, GLuint *index*);
void **glDisablei**(GLenum *cap*, GLuint *index*);

Parameters
cap:
Specifies a symbolic constant indicating a GL capability.
index
Specifies the index of the switch to disable (for **glEnablei** and **glDisablei** only).

Using the Depth Buffer for Hidden surface Removal

1. Color Buffers

The color buffers are the ones to which you usually draw. They contain RGB color data and may also contain alpha values. Double-buffered systems have front and back buffers, and a single-buffered system has the front buffers only.

2. Depth Buffer

The depth buffer stores a depth value for each pixel. Depth is usually measured in terms of distance to the eye, so pixels with larger depth-buffer values are overwritten by pixels with smaller values. This is just a useful convention, however, and the depth buffer's behavior can be modified as described in **"Depth Test."** The depth buffer is sometimes called the **z buffer** (the z comes from the fact that x and y values measure horizontal and vertical displacement on the screen, and the z value measures distance perpendicular to the screen).

Clearing Buffers

The following commands set the clearing values for each buffer.

i. **Void glClearColor(r,g,b,a);**
ii. **void glClearDepth(depth);**

After you've selected your clearing values and you're ready to clear the buffers, using **glClear().**

Void glClear*(*GLbitfieldmask*);*

This Clears the specified buffers. The value of mask is the bitwise logical OR of some combination of **GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT,** to identify which buffers are to be cleared.

For example:

```
glClearDepth(1.f);
glClearColor(0.3f, 0.3f, 0.3f, 0.f); //background colour
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Testing and Operating on Fragments

Alpha Test

In RGBA mode, the alpha test allows you to accept or reject a fragment based on its alpha value. The alpha test is enabled and disabled by passing **GL_ALPHA_TEST** to **glEnable()** and **glDisable().** To determine whether the alpha test is enabled, use **GL_ALPHA_TEST** with

glIsEnabled().

If enabled, the test compares the incoming alpha value with a reference value. The fragment is accepted or rejected depending on the result of the comparison. Both the reference value and the comparison function are set with **glAlphaFunc()**.

Void glAlphaFunc(GLenum func, Glclampf ref);
Sets the reference value and comparison function for the alpha test. The reference value ref is clamped to be between zero and one.

Depth Test

For each pixel on the screen, the depth buffer keeps track of the distance between the viewpoint and the object occupying that pixel. Then if the specified depth test passes, the incoming depth value replaces the one already in the depth buffer.

The depth buffer is generally used for hidden-surface elimination. If a new candidate color for that pixel appears, it's drawn only if the corresponding object is closer than the previous object. In this way, after the entire scene has been rendered, only objects that aren't obscured by other items remain. Initially, the clearing value for the depth buffer is a value that's as far from the viewpoint as possible, so the depth of any object is nearer than that value. If this is how you want to use the depth buffer, you simply have to enable it by passing **GL_DEPTH_TEST** to **glEnable()** and remember to clear the depth buffer before you redraw each frame.

Enabling Depth Checking

1. glEnable(GL_DEPTH_TEST);
If enabled, do depth comparisons and update the depth buffer. Note that even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled.
2. glDepthMask(GL_TRUE);

glDepthMask specifies whether the depth buffer is enabled for writing. If flag is **GL_FALSE**, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Depth Function
You can also choose a different comparison function for the depth test with **glDepthFunc().**
Void glDepthFunc(GLenum func);
Sets the comparison function for the depth test. The value for func must be **GL_NEVER,**

GL_ALWAYS, **GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER**,

or **GL_NOTEQUAL**

. An incoming fragment passes the depth test if its z value has the specified relation to the value already stored in the depth buffer. The default is **GL_LESS**, which means that an incoming fragment passes the test if its z value is less than that already stored in the depth buffer. In this case, the z value represents the distance from the object to the viewpoint, and smaller values mean the corresponding objects are closer to the viewpoint.

Note: Study the rotation of color cube program under this topic which can be considered as a possible question under this topic

Programmable Shaders :

GLSL (GLslang) is a short term for the official OpenGL Shading Language. GLSL is a C/C++ similar high level programming language for several parts of the graphic card. With GLSL you can code (right up to) short programs, called shaders, which are executed on the GPU.

There are two types of shaders in GLSL: vertex shaders and fragment shaders.

Vertex Shader:

A vertex shader operates on every vertex. So if you call glVertex* (or glDrawArrays, …) the vertex shader is executed for each vertex. If you use a vertex shader you have nearly full control over what is happening with each vertex. But if you use a vertex shader ALL Per-Vertex operations of the fixed function OpenGL pipeline are replaced.

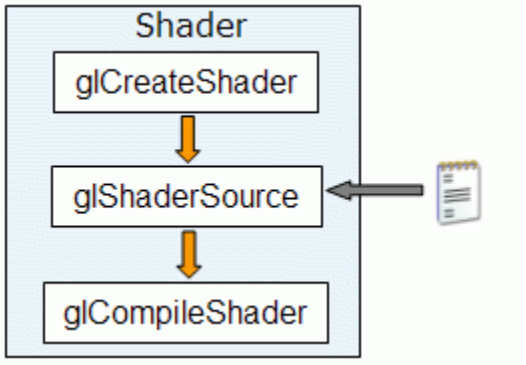
Fragment Shader:

A fragment shader operates on every fragment which is produced by rasterization. With fragment shader you have nearly full control over what is happening with each fragment. But just like a vertex shader, a fragment shader replaces ALL Per-Fragment operations of the fixed function OpenGL pipeline.

The vertex processor is responsible for running the vertex shaders. The fragment processor is where the fragment shaders run.

OpenGL Setup for GLSL - Creating a Shader

The following figure shows the necessary steps to create a shader.



The first step is creating an object which will act as a shader container. The function available for this purpose returns a handle for the container.

Syntax:

GLuint glCreateShader(GLenum shaderType);

Parameter:

shaderType - GL_VERTEX_SHADER or GL_FRAGMENT_SHADER.

The following step is to add some source code. The source code for a shader is a string array, although you can use a pointer to a single string.
Syntax for a shader is:

void glShaderSource(GLuint shader, int numOfStrings, const char **strings, int *lenOfStrings);

Parameters:
shader - the handler to the shader.
numOfStrings - the number of strings in the array.
strings - the array of strings.
lenOfStrings - an array with the length of each string, or NULL, meaning that the strings are NULL terminated.

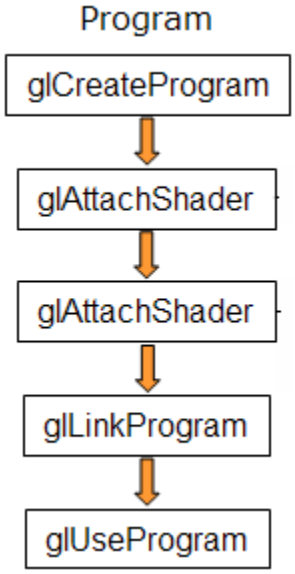
Finally, the shader must be compiled. The function to achieve this is:

void glCompileShader(GLuint shader);

Parameters:
shader - the handler to the shader.

OpenGL Setup for GLSL - Creating a Program:

The following figure shows the necessary steps to get a shader program ready and going.



The first step is creating an object which will act as a program container. The function available for this purpose returns a handle for the container.

The syntax for this function,is as follows:

GLuint glCreateProgram(void);

You can create as many programs as you want. Once rendering, you can switch from program to program, and even go back to fixed functionality during a single frame. For instance you may want to draw a teapot with refraction and reflection shaders, while having a cube map displayed

for background using OpenGL's fixed functionality.

The next step involves attaching the shaders created in the previous subsection to the program you've just created. The shaders do not need to be compiled at this time; they don't even have to have source code. All that is required to attach a shader to a program is the shader container.

To attach a shader to a program use the following function:

void glAttachShader(GLuint program, GLuint shader);

Parameters:

program - the handler to the program.
shader - the handler to the shader you want to attach.

If you have a pair vertex/fragment of shaders you'll need to attach both to the program. You can have many shaders of the same type (vertex or fragment) attached to the same program, just like a C program can have many modules. For each type of shader there can only be one shader with a main function, also as in C.

You can attach a shader to multiple programs, for instance if you plan to use the same vertex shader in several programs.

The final step is to link the program. In order to carry out this step the shaders must be compiled as described in the previous subsection.

The syntax for the link function,is as follows:

void glLinkProgram(GLuint program);

Parameters:

program - the handler to the program.

After the link operation the shader's source can be modified, and the shaders recompiled without affecting the program.

As shown in the figure above, after linking the program, there is a function to actually load and use the program (OpenGL 2.0) glUseProgram. Each program is assigned an handler, and you can have as many programs linked and ready to use as you want (and your hardware allows).

The syntax for this function is as follows:

void glUseProgram(GLuint prog);

Parameters:

prog - the handler to the program you want to use, or zero to return to fixed functionality

If a program is in use, and it is linked again, it will automatically be placed in use again, so in this case you don't need to call this function again. I the parameter is zero then the fixed functionality is activated.

OpenGL Setup for GLSL - Example

```
void setShaders()
{
		char *vs,*fs;

		v = glCreateShader(GL_VERTEX_SHADER);
		f = glCreateShader(GL_FRAGMENT_SHADER);

		vs = textFileRead("toon.vert");
		fs = textFileRead("toon.frag");

		const char * vv = vs;
		const char * ff = fs;

		glShaderSource(v, 1, &vv,NULL);
		glShaderSource(f, 1, &ff,NULL);

		free(vs);free(fs);

		glCompileShader(v);
		glCompileShader(f);

		p = glCreateProgram();

		glAttachShader(p,v);
		glAttachShader(p,f);

		glLinkProgram(p);
		glUseProgram(p);
}
```

The InfoLog:

The status of the compile steps can be queried with the following function:

void glGetShaderiv(GLuint object, GLenum type, int *param);

Parameters:
object - the handler to the object. Either a shader or a program
type - GL_COMPILE_STATUS.
param - the return value, GL_TRUE if OK, GL_FALSE otherwise.

The status of the link step can be queried with the following function:

void glGetProgramiv(GLuint object, GLenum type, int *param);

Parameters:
object - the handler to the object. Either a shader or a program
type - GL_LINK_STATUS.
param - the return value, GL_TRUE if OK, GL_FALSE otherwise.

In order to get the InfoLog for a particular shader or program use the following functions:

void glGetShaderInfoLog(GLuint object, int maxLen, int *len, char *log);
void glGetProgramInfoLog(GLuint object, int maxLen, int *len, char *log);

Parameters:

object - the handler to the object. Either a shader or a program
maxLen - The maximum number of chars to retrieve from the InfoLog.
len - returns the actual length of the retrieved InfoLog.
log - The log itself.

GLSL has two types of variable qualifiers:

1) **Uniform:** Uniforms are values which do not change during a rendering, for example the light position or the light color. Uniforms are available in vertex and fragment shaders. Uniforms are read-only.
2) **Attribute:** Attributes are only available in vertex shader and they are input values which change every vertex, for example the vertex position or normals. Attributes are read-only.

Uniform Variables
The function to retrieve the location of an uniform variable given its name, as defined in the shader, is:

GLint glGetUniformLocation(GLuint program, const char *name);

Parameters:
program - the handler to the program
name - the name of the variable.

The return value is the location of the variable, which can then be used to assign values to it. A

family of functions is provided for setting uniform variables, its usage being dependent on the data type of the variable. A set of functions is defined for setting float values as:

void glUniform1f(GLint location, GLfloat v0);
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);

or

GLint glUniform{1,2,3,4}fv(GLint location, GLsizei count, GLfloat *v);

Parameters:
location - the previously queried location.
v0,v1,v2,v3 - float values.
count - the number of elements in the array
v - an array of floats.

A similar set of function is available for data type integer, where "f" is replaced by "i". There are no functions specifically for bools, or boolean vectors. Just use the functions available for float or integer and set zero for false, and anything else for true.

Example for Uniform variable:

```
GLint loc1,loc2,loc3,loc4;
  float specIntensity = 0.98;
  float sc[4] = {0.8,0.8,0.8,1.0};
  float threshold[2] = {0.5,0.25};
  float colors[12] = {0.4,0.4,0.8,1.0,
                      0.2,0.2,0.4,1.0,
                      0.1,0.1,0.1,1.0};

  loc1 = glGetUniformLocation(p,"specIntensity");
  glUniform1f(loc1,specIntensity);

  loc2 = glGetUniformLocation(p,"specColor");
  glUniform4fv(loc2,1,sc);

  loc3 = glGetUniformLocation(p,"t");
  glUniform1fv(loc3,2,threshold);

  loc4 = glGetUniformLocation(p,"colors");
  glUniform4fv(loc4,3,colors);
```

Attribute Variables:

For uniform variables, it is necessary to get the location in memory of the variable
GLint glGetAttribLocation(GLuint program,char *name);
Parameters:
program - the handle to the program.
name - the name of the variable

The variable's location in memory is obtained as the return value of the above function. The next step is to specify a value for it, potentially per vertex. As in the uniform variables, there is a function for each data type.

void glVertexAttrib1f(GLint location, GLfloat v0);
void glVertexAttrib2f(GLint location, GLfloat v0, GLfloat v1);
void glVertexAttrib3f(GLint location, GLfloat v0, GLfloat v1,GLfloat v2);
void glVertexAttrib4f(GLint location, GLfloat v0, GLfloat v1,,GLfloat v2, GLfloat v3);

or

GLint glVertexAttrib{1,2,3,4}fv(GLint location, GLfloat *v);

Parameters:
location - the previously queried location.
v0,v1,v2,v3 - float values.
v - an array of floats.

Example:

```
glBegin(GL_TRIANGLE_STRIP);

        glVertexAttrib1f(loc,2.0);
        glVertex2f(-1,1);

        glVertexAttrib1f(loc,2.0);
        glVertex2f(1,1);

        glVertexAttrib1f(loc,-2.0);
        glVertex2f(-1,-1);

        glVertexAttrib1f(loc,-2.0);
        glVertex2f(1,-1);

    glEnd();
```